Advanced Software Fault Tolerance Strategies for Mission Critical Spacecraft Applications

for
NASA Ames Research Center IV&V Facility
Software Initiative UPN 323-08

Task 4 Report
September 30, 1999
Recommended Flight Software Development Methodology, Activities, and Architecture

Prepared by JSC NX Technology Division

Charles Nagy
Paul Porter
Richard Gilreath

Approved by M. Himel/Chief, NX Technology Division

Purpose and Scope

This is the fourth paper in a series of four papers. A review of the scope and findings of each of the preceding papers will place the scope and recommendations of this paper in perspective.

The purpose of the first paper in this series was to provide a survey of empirically validated techniques for generating software fault-tolerant systems or extremely reliable software systems. A second objective was to determine if any conceptually new approaches had been developed or validated since the appearance of the original fault tolerance approaches of N-version programming and recovery. An extensive literature search to identify methods, tools, and techniques claiming to provide either highly reliable or fault-tolerant software systems led to the following findings:

1. N-version and recovery still form the basis of all software fault-tolerant systems.
2. Formal methods show great promise but are hampered by several drawbacks:
   - Knowledge required is usually not part of the computer science curriculum;
   - Unable to identify incomplete requirements; and
   - Very expensive to fully implement.
3. The use of assertions can be quite powerful and is within the grasp of almost all developers.
4. Model state checking is a strong approach to identifying incomplete requirements.
5. The real world will require a combination of approaches, for example:
   The distributed recovery block method is designed around the use of try/retry blocks, acceptance tests, voter mechanisms, and object oriented coding. Given these elements it would seem logical that critical functions could be identified (using fault trees etc) for incorporation into objects which would be independently developed (as in n-version) and run on different processors. Voting mechanisms and acceptance tests could then be used to validate the outputs and determine which data to act on. Non-critical functions could be handled in a conventional manner. This grants the benefits of n-version for the most critical functions and limits the costs.

The purpose of the second paper was to analyze and categorize documented software anomalies encountered in real-time control systems. The majority of data collected was from the National Space Transportation System (Space Shuttle) program, but information was also collected for some "well-known" failures induced by software. As a result of that effort, the following findings were presented:

1. Data related to software induced failures are not easily acquired;
2. Incomplete or inconsistent requirements can impose inordinate costs on a program; and
3. Exhaustive testing fails to prevent errors in the operations environment.

The purpose of the third paper was to examine software fault tolerance and reliability methods used in the evolving X-38 201 flight software development project. Beyond simply identifying those methods currently being applied, this paper suggested opportunities for applying other methods and for enhancing those methods currently being used to improve safety and reliability in X-38 software. The challenges posed by the architecture and the software development approach used for X-38 flight software, as well as the opportunities for improvement, will not be presented in this paper. For a detailed discussion of these topics, please refer to the appropriate sections of the third paper. However, in the context of examining the broader implications of design decisions, the following findings of the third paper are presented:

1. Consistent with one of the findings in the first report, a combination of methods is used in the X-38 201 software to handle faults. These include fault tolerance methods and fault evasion methods. Some methods identified in the first report are not applied.

2. Most of the fault tolerance and fault evasion methods applied in the X-38 201 software are *in theory* applicable to both hardware and software faults. However, the architecture focuses on detecting and recovering from hardware faults, as demonstrated by numerous requirements.
3. Classifying software faults broadly as either timing or data faults, the methods that address software faults focus on handling timing faults, which can be caused by either hardware or software. These include output-buffer-not-empty, input-buffer-not-full, and task (processor budget) overruns.
4. Most of the methods being applied to handle faults are implemented in the FTSS [Fault Tolerant System Services] software (which is consistent with the architectural focus on hardware faults).
5. N-versioning, identified in the first report as being a cornerstone in software fault-tolerant systems, is not used at any level. The use of N-versioning is precluded by the architecture, which requires replicated software in all functionally identical processors.

The purpose of this final paper is to integrate the findings of the first three papers and to provide recommendations for flight software development methodology, activities, and architecture. In support of this paper, an extensive literature search was also conducted. However, the search focused on finding fault-tolerant software examples where the various techniques were applied and on identifying software development standards and methodologies. The sources consulted are listed in Appendix A.

Recommendations

Although not all reliable software is safety-critical, it is true that all safety-critical software must be reliable. Therefore, the recommendations will address software development methodologies, activities, and architecture whose purpose is to increase software reliability as a necessary precursor to software fault tolerance as well as those whose purpose is to provide fault tolerance.

Although some activities to improve reliability and safety of a flight system design can be implemented later in the development cycle, activities targeted for the beginning of the development cycle will produce the greatest positive impact on the safety and reliability of the developed system. Therefore, the following recommendations focus on activities that should be performed early in the system development cycle.

At the beginning of flight system development, the following activities are recommended:

1. Conduct research of technical literature as a first step in a conceptual design process.
2. Identify and evaluate software standards for applicability.
3. Identify and evaluate software fault-tolerant architectures and methodologies.
4. Identify all requirements (*e.g.*, performance, safety) for the target system.
5. Identify tools that can be used to demonstrate software fault tolerance.

When making design decisions, the following recommendations should be considered:

6. Select a system design (hardware and software) whose complexity does not preclude demonstrating, within budget and schedule, that all design requirements have been met.
7. Select a software development standard and methodology that insures the highest level of reliability and safety.
8. Select a software development package (implemented language, compiler, and test tools) that will provide a development environment which automates as many required functions as possible.

9. If tools to demonstrate software fault tolerance for the candidate design are not available, design, code, test, and validate these tools as part of the software project.

The rationale for each of these recommendations will be presented in the following section.

Discussion of Recommendations

The goal of flight software development is to produce a flight system (software and hardware), within budget and schedule, that performs the functions intended and that subjects crew and equipment to both an acceptable and a demonstrable level of risk. With this in mind, all recommendations are made to minimize risk as well as to provide the required functionality.

It is also important to note that software fault tolerance, although necessary, is not sufficient to insure minimization of risk. Fault tolerance can, and should, be applied at three levels: hardware, software, and system (interfaces). All three levels are susceptible to design, implementation, or maintenance errors. Traditionally, redundant communications, replicated processors, additional memory, and redundant power supplies are among the measures used to provide hardware fault tolerance. The measures used to provide software fault tolerance were identified in the first paper and included mechanisms such as checkpoint/restart, recovery blocks, and N-version programming. System fault tolerance recognizes that software and hardware do not exist independently, that they execute in some environment, and that operational interfaces, human or computer, can lead to the introduction of faults into the system.

1. Conduct research of technical literature as a first step in a conceptual design process.

In a growing environment of "faster, better, cheaper," it is imperative that project management seeks the benefit of the experience of other software developers with respect to system designs; software standards and methodologies; and software verification and validation techniques. The benefits of this research extend beyond the realms of cost and schedule into the arena of reliability, safety, and minimization of risk.

As reported in the first paper, N-version and recovery still form the basis of all software fault-tolerant systems. However, other designs aimed at achieving some degree of software fault tolerance can be found in current technical literature. Some designs have had limited success while others were unsuccessful. In either case, the literature documents the problems encountered and the solutions attempted. For example, the problems associated with Byzantine agreements (voting algorithms) and with multiple time-synchronized processes can be found in several technical reports.

In addition to NASA sources such as the NASA Independent Verification and Validation (IV&V) Facility and Langley Research Center, applicable technical literature can be obtained from other government sources such as the Department of Defense (DoD) Software Engineering Institute (SEI) as well as universities and contractors. The list of resources consulted in Appendix A provides a glimpse of the variety of sources available.

2. Identify and evaluate software standards for applicability.

One of the purposes of software standards is to identify activities and processes that have been empirically verified as elements of successful software development projects. Software standards are not static but have evolved to take into account demonstrable software process improvements. For example, the DoD adopted DOD-STD-2167A in February 1988, changed to MIL-STD-498 in December 1994, and then changed to IEEE/EIA 12207 in March 1998.

During the literature search for this paper, many software standards were found. In addition to the standards cited above, the following NASA documents were found:

- NASA/TP-98-208193, Release 2.0. "Formal Methods Specification and Verification Guidebook for Software and Computer Systems," NASA Office of Safety and Mission Assurance, December, 1998.
- NASA-GB-1740.13-96. "NASA Guidebook for Safety Critical Software – Analysis and Development," National Aeronautics and Space Administration, April, 1996.
- NASA-STD-8719.13A. "Software Safety," NASA Technical Standard, September 15, 1997.
- NPD 2820.1. "NASA Software Policies," NASA Policy Directive, May 29, 1998.

Additionally, the Federal Aviation Administration (FAA) uses a regulated standard for certification of airborne software systems, RTCA/DO-187B, "Software Considerations In Airborne Systems And Equipment Certification." Since software is present in many home appliances, Underwriters Laboratory also has a software standard.

Project management should evaluate each standard from the perspective of how well the processes specified in the standard minimize the management risk associated with both the development of safety-critical software and the risk to crew and equipment associated with the developed software.

3. Identify and evaluate software fault-tolerant architectures and methodologies.

In the first paper, several empirically validated techniques for generating software fault-tolerant systems or extremely reliable software systems were presented. The literature search that produced that list should be performed by project management not only to determine which techniques are available but also to determine the strengths and weaknesses of each approach.

4. Identify all requirements (*e.g.*, performance, safety) for the target system.

To preclude rework and to minimize risk, all requirements for the target system should be identified as early in the development cycle as possible. These include not only performance requirements but also the safety requirements that are necessary to assure that the target system does not pose an unacceptable risk to crew and equipment.

In addition to the documents cited above, the following NASA documents addressing safety requirements should be considered for applicability:

- JSC-28354. "Human-Rating Requirements," National Aeronautics and Space Administration, Johnson Space Center, June 1998.
- NSTS 1700.7B. "Safety Policy and Requirements for Payloads Using the Space Transportation System," National Aeronautics and Space Administration, Johnson Space Center, October 12, 1998.
- NSTS 1700.7B ISS Addendum. "Safety Policy Requirements for Payloads Using the International Space Station," National Aeronautics and Space Administration, Johnson Space Center, December 8, 1995.
- SSP 30309, Revision E. "Safety Analysis and Risk Assessment Requirements Document," National Aeronautics and Space Administration, International Space Station Alpha Program, Johnson Space Center, October 28, 1994.

- SSP 50038, Revision B. "Computer-Based Control System Safety Requirements," National Aeronautics and Space Administration, International Space Station Alpha Program, Johnson Space Center, November 17, 1995.

5. Identify tools that can be used to demonstrate software fault tolerance.

One of the fundamental requirements for every software development effort is to verify that the developed system has the capabilities specified in the design requirements. For safety critical software that is claimed to be fault-tolerant, the developer has to demonstrate not only that the developed system satisfies its design requirements in nominal operation without faults but also that the developed system satisfies these requirements in the presence of faults. Since the correct operation of fault-tolerant mechanisms is essential to support the claimed risk minimization to crew and equipment, the demonstration of correct operation of these mechanisms is mandatory.

To demonstrate the correct operation of fault-tolerant mechanisms, several tool types have been identified. For example, there are tools that improve reliability by measuring test coverage (*e.g.*, CodeTEST) and there are tools based on a "fault injection" methodology to directly test the operation of fault-tolerant mechanisms.

Project management should identify candidate tools to be used to demonstrate software fault tolerance as early in the development cycle as possible.

6. Select a system design (hardware and software) whose complexity does not preclude demonstrating, within budget and schedule, that all design requirements have been met.

Using the information obtained through the technical literature research activity, project management should evaluate candidate system designs and select the target system design based upon at least these criteria:

- Will the system minimize risk to crew and equipment to an acceptable level?
- Can the fault-tolerant capabilities (hardware, software and system) be demonstrated?
- Can the required functional capabilities of the system be demonstrated?
- Can all elements of system development be completed within budget and schedule?

If this evaluation is not performed, the risk to crew and equipment posed by the developed system as well as the risk of not completing all required elements of software development is increased.

7. Select a software development standard and methodology that insures the highest level of reliability and safety.

Most standards contain a rigorous, structured methodology for the development of software systems. The use of a structured methodology assures that all activities and processes required for the successful completion of software development projects are identified at the beginning of the development cycle.

If the goal of the project is to develop safety critical software, it is imperative that a structured approach be used. Only by employing the technical rigor of a structured approach will the project be able to demonstrate that the developed system performs the functions intended and exposes crew and equipment to an acceptable level of risk.

The use of a prototyping methodology ("design a little, test a little, repeat") for the development of safety critical software is not recommended. In fact, none of the resources consulted recommended that prototyping be used to develop safety critical software.

Although prototyping has been successfully used to develop software that is not safety critical, the basis of the method is antithetical to the production of safety critical software, which must be extremely reliable or software fault-tolerant. For safety critical software, this "design a little, test a little, repeat" methodology poses the following concerns.

1. When using this approach, the requirements specification document is generally considered a "living" document, which is modified based upon test results and remains modifiable throughout the software development cycle. At the end of the development cycle, this document may only reflect design requirements discovered through testing in a test environment and not reflect the design requirements necessary to support its operation in the target environment.

2. The methodology assumes that testing alone will result in the successful production of safety critical software. Even if the testing were exhaustive, one of the findings of the second paper was that exhaustive testing fails to prevent errors in the operations environment.

3. Although it is claimed that the method reduces program costs, use of the method to develop safety critical software may cost as much or more than structured methods. Since the methodology emphasizes testing rather than requirement definition, the finding in the second paper, that incomplete or inconsistent requirements can impose inordinate costs on a program, applies.

4. Under budget and schedule constraints, the "design a little, test a little, repeat" methodology may degenerate into a "find a hole, plug a hole" methodology where completion of the project on schedule and within budget become primary goals and where safety and risk minimization become secondary.

5. Under budget and schedule constraints, the results of testing can drive the design rather than the design driving the testing. When design changes are incorporated late in the development cycle, both the costs and the risks associated with the developed system increase.

A characteristic of safety critical flight software should be that it minimizes risk to crew and equipment to an acceptable and demonstrable level. The use of a prototyping methodology is not consistent with producing software having this characteristic.

Structured methodologies were developed decades ago to preclude the accidents and failures associated with using prototyping methodologies for safety critical systems. Project management should select a proven methodology for the development of safety critical software.

8. Select a software development package (implemented language, compiler, and test tools) that will provide a development environment which automates as many required functions as possible.

The selection of the programming language to be used can affect the reliability of the developed system. A programming language is usually implemented in a software development package containing a compiler and a set of tools. Even if the same underlying language is used, each software development package has its strengths and weaknesses in providing an environment for the development of extremely reliable or safety critical software.

Consider the following categorization of software errors derived from the descriptions of 342 failures of products found during acceptance test or operation. The data are located in the Fault & Failure Analysis Repository (EFF) of the National Institute of Standards and Technology (NIST) and the categorization was provided by NIST.

- Calculation Errors
- Initialization Errors
- Timing Errors
- Interface Errors
- Change Impact Errors
- Omission Errors
- Configuration Management Errors
- Data Errors
- Requirements Errors
- Quality Assurance Errors
- Logic Errors
- Fault Tolerance Errors
- Other Errors

Many of the errors described in these categories can be automatically identified at compile time because the rules of the associated programming language preclude specific error types.

For example, one candidate programming language that should be considered for safety critical applications is Ada. Several industry surveys indicate that Ada is a preferred language for the development of safety critical software. Ada has been successfully used in defense systems, banking and information systems, passenger railroad systems, commercial aviation, communications systems, biomedical applications, and vehicle based software. One reason given for the use of an Ada software development environment is that many errors (*e.g.*, initialization errors, data errors, interface errors) are automatically identified by the Ada compiler. By automatically identifying certain types of errors, the compiler reduces test time required and contributes to the reliability of the software product.

Also, many software development packages contain tools to track software revisions and provide configuration management capabilities. Such tools can be used to minimize change impact and configuration management errors.

Project management should evaluate candidate software development packages early in the development cycle and select one that automates as many required functions as possible.

9. <u>If tools to demonstrate software fault tolerance for the candidate design are not available, design, code, test, and validate these tools as part of the software project.</u>

As systems become more complex, the tools required to demonstrate system capabilities tend to become more complex. If a design is selected that is "state-of-the-art" or represents "leading-edge technology," it is very likely that conventional (existent) tools will not be adequate to demonstrate the capabilities of the selected design.

As stated earlier, the demonstration of correct operation of fault-tolerant mechanisms in the developed system is mandatory. If existent tools that demonstrate the correct operation of these mechanisms cannot be identified, project management should plan to develop these tools as part of the project and adjust budget and schedule to account for this activity.

Appendix A

Resources Consulted

Internet Links to Related Documents

(DO-178b) Software Considerations in Airborne Systems and Equipment Certification SC-167
http://www.cs.cmu.edu/People/Koopman/depend/book/rtca92.htm
29th International Fault-Tolerant Computing Symposium
http://chaos.crhc.uiuc.edu/FTCS-29/fastabs.html
Ada Information Clearinghouse – AdaIC
http://www.adaic.org/
AI Lab Zurich Links Embedded and Real-Time Systems
http://www.ifi.unizh.ch/groups/ailab/links/embedded.html
Bell Laboratories Computing and Mathematical Science
http://netlib.bell-labs.com/
Charles Stark Draper Laboratory
http://www.draper.com/
CodeTEST--Embedded Software Test and Analysis Tools
http://www.amc.com/products/embedded_sw_test.html
Conceptual Systems & Software (CSS) Publications
http://www.consys.com/publications/vv_test.html
Department of Defense Office of Technology Transition
http://www.dtic.mil/techtransit/index.html
DO-178B, Software Considerations in Airborne Systems and Equipment Certification - Oct 98
http://www.stsc.hill.af.mil/crossTalk/1998/oct/schad.asp
Draft Software Fault and Failure Handbook
http://hissa.ncsl.nist.gov/effProject/handbook/index.html
Fault-Tolerant Computing Systems
http://www.computer.org/conferen/proceed/FTCS95/FTCS95TC.HTM
Fermilab Publications Office Home Page
http://fnalpubs.fnal.gov/techpubs/
Formal Verification for Fault-Tolerant Architectures Some Lessons Learned
http://www.csl.sri.com/reports/html/fme93.html
IEEE-EIA 12207 Description
http://www.software.org/Quagmire/descriptions/us-12207.html
Introduction to VxWorks
http://fst.jpl.nasa.gov/usersguide/vxintro.html
ISO-IEC 12207 Software Lifecycle Processes - Aug 1996
http://www.stsc.hill.af.mil/crosstalk/1996/aug/isoiec.asp
Langley Technical Report Server
http://techreports.larc.nasa.gov/ltrs/abs.html
Lucent Technologies' NT SwiFT
http://www.bell-labs.com/projects/swift/
Mathematical Systems Theory, Volume 26
http://www.cse.unsw.edu.au/dblp/db/journals/mst/mst26.html
Military Specifications
http://www.aeroflex.com/act/mil-spec.htm
MIL-STD-498
http://www-library.itsi.disa.mil/mil_std/std498.html
Minas Project-Byzantine Agreement
http://www.cs.bham.ac.uk/teaching/examples/simjava/byzantine-agreement/byzant_agree.html

NASA Guidebooks and Standards
http://www.ivv.nasa.gov/SWG/resources/
NASA IV&V Facility Homepage
http://www.ivv.nasa.gov/
NASA NODIS Directives Library Index
http://nodis.hq.nasa.gov/Library/loglib.html
NASA Software Research Laboratory (SRL) Database
http://research.ivv.nasa.gov/docs/techreports.html
NASA/WVU Software Research Laboratory Home Page
http://research.ivv.nasa.gov/
Real-Time Systems
http://www.cs.umd.edu/~fwmiller/etc/realtime.html
Relex Software Corporation - Reliability Prediction
http://www.innovsw.com/index.htm
Reliable Software Technologies -- Software Assurance
http://www.rstcorp.com/
Safety-Critical Systems Computer Language Survey - Results
http://www.comlab.ox.ac.uk/archive/safety/lang-survey.html
Scope for Underwriter's Laboratory UL 1998_1
http://ulstandardsinfonet.ul.com/scopes/1998.html
Selected Real-Time Computing Resources
http://www.realtime-os.com/rtresour.html
SofSys, LLC - Analysis of fault-tolerant control systems
http://www.sof-sys.com/
Software Engineering Institute (SEI) Document List
http://www.sei.cmu.edu/publications/documents/doc.list/
Software Engineering Institute (SEI) Document List
http://www.sei.cmu.edu/publications/documents/doc.list/
Software Engineering Institute (SEI) Home Page
http://www.sei.cmu.edu/
Software Engineering Research Laboratory
http://www.cs.colorado.edu/users/serl/
Software Standards
http://sepo.nosc.mil/498.html
SoHaR - Software and Hardware Reliability Web Site
http://fermi.sohar.com/
The World Wide Web Virtual Library Formal Methods
http://www.comlab.ox.ac.uk/archive/formal-methods.html
The World Wide Web Virtual Library Safety-Critical
http://www.comlab.ox.ac.uk/archive/safety.html
The WWW Virtual Library Computing
http://src.doc.ic.ac.uk/bySubject/Computing/Overview.html
WWW Virtual Library - Software Engineering
http://ricis.cl.uh.edu/virt-lib/soft-eng.html
xSuds Software Understanding System
http://xsuds.bellcore.com/

Documents Reviewed

Arrowsmith, Beth, and McMillin, Bruce. "How to Program in CCSP," Department of Computer Science, University of Missouri-Rolla, CSC 94-20, August 30, 1994.

Arrowsmith, Beth, and McMillin, Bruce. "Teaching the Practice of Formal Methods in Distributed Computing Systems – A Module," Department of Computer Science, University of Missouri-Rolla, CSC 94-19, July 30, 1994.

Arrowsmith, Beth; McMillin, Bruce; and Wilkerson, Ralph. "Formal Methods: How, When, and Why They are Used," Department of Computer Science, University of Missouri-Rolla, CSC 95-10, October, 1995.

Brackett, John W. "Software Requirements," Software Engineering Institute, Carnegie Mellon University, SEI Curriculum Module SEI-CM-19-1.2, January, 1990.

Butler, Ricky W., and Finelli, George B. "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," NASA Langley Research Center.

Christel, Michael G., and Kang, Kyo C. "Issues in Requirements Elicitation," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-92-TR-12, September, 1992.

Ferguson, Jack R., and DeRiso, Michael E. "Software Acquisition: A Comparison of DoD and Commercial Practices," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-94-SR-9, October, 1994.

Gong, Li, and Goldberg, Jack. "Implementing Adaptive Fault-Tolerant Services for Hybrid Faults," Computer Science Laboratory, SRI International, September 30, 1994.

Gong, Li; Lincoln, Patrick; and Rushby, John. "Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults," Computer Science Laboratory, SRI International, September, 1995.

Gray, Lewis. "ISO/IEC 12207 Software Lifecycle Processes," Ada PROS, Inc., August, 1996.

Heimerdinger, Walter L., and Weinstock, Charles B. "A Conceptual Framework for System Fault Tolerance," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-92-TR-33, October, 1992.

Heimerdinger, Walter L., and Weinstock, Charles B. "Fault-tolerant Systems Practitioner's Workshop, June 10-11, 1991," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-91-SR-13, October, 1991.

Holloway, C. Michael. "Why Engineers Should Consider Formal Methods," NASA Langley Research Center, October, 1997.

Hook, Audrey A.; Brykczynski, Bill; McDonald, Catherine W.; Nash, Sarah H.; and Youngblut, Christine. "A Survey of Computer Programming Languages Currently Used in the Department of Defense," Institute for Defense Analyses, June 15, 1995.

IEEE/EIA 12207.0-1996. "(ISO/IEC 12207) Standard for Information Technology – Software life cycle processes," The Institute of Electrical and Electronics Engineers, Inc., March, 1998.

IEEE/EIA 12207.1-1997. "(ISO/IEC 12207) Standard for Information Technology – Software life cycle processes – Life cycle data," The Institute of Electrical and Electronics Engineers, Inc., April, 1998.

IEEE/EIA 12207.2-1997. "(ISO/IEC 12207) Standard for Information Technology – Software life cycle processes – Implementation considerations," The Institute of Electrical and Electronics Engineers, Inc., April, 1998.

Johnson, Leslie A. (Schad). "DO-178B, 'Software Considerations in Airborne Systems and Equipment Certification'," Boeing Commercial Airplane Group, October, 1998.

JSC-28354. "Human-Rating Requirements," National Aeronautics and Space Administration, Johnson Space Center, June 1998.

Lane, Thomas G. "Studying Software Architecture Through Design Spaces and Rules," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-90-TR-18, November, 1990.

Lincoln, Patrick, and Rushby, John. "A Formally Verified Algorithm for Interactive Consistency Under A Hybrid Fault Model," Computer Science Laboratory, SRI International, June, 1993.

Lutfiyya, Hanan; McMillin, Bruce; Arrowsmith, Beth; and Serban, Cristina. "CCSP – A Formal System for Distributed Program Debugging," Department of Computer Science, University of Missouri-Rolla, UMR Computer Science Technical Report Number 94-30, 1994.

McGarry, Frank; Pajerski, Rose; Page, Gerald; and Waligora, Sharon. "Software Process Improvement in the NASA Software Engineering Laboratory," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-94-TR-22, December, 1994.

MIL-STD-498. "Software Development and Documentation," Department of Defense, December 5, 1994.

Moore, Jim. "ISO 12207 and Related Software Life-Cycle Standards," The MITRE Corporation.

NASA/TP-98-208193, Release 2.0. "Formal Methods Specification and Verification Guidebook for Software and Computer Systems", NASA Office of Safety and Mission Assurance, December, 1998.

NASA-GB-1740.13-96. "NASA Guidebook for Safety Critical Software – Analysis and Development," National Aeronautics and Space Administration, April, 1996.

NASA-STD-8719.13A. "Software Safety," NASA Technical Standard, September 15, 1997.

NPD 2820.1. "NASA Software Policies," NASA Policy Directive, May 29, 1998.

NSTS 1700.7B ISS Addendum. "Safety Policy Requirements for Payloads Using the International Space Station," National Aeronautics and Space Administration, Johnson Space Center, December 8, 1995.

NSTS 1700.7B. "Safety Policy and Requirements for Payloads Using the Space Transportation System," National Aeronautics and Space Administration, Johnson Space Center, October 12, 1998.

Owre, Sam; Rushby, John; Shankar, Natarajan; and von Henke, Friedrich. "Formal Verification for Fault-Tolerant Architectures: Some Lessons Learned," Computer Science Laboratory, SRI International, April, 1993.

Place, Patrick R.H., and Kang, Kyo C. "Safety-Critical Software: Status Report and Annotated Bibliography," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-92-TR-5, June, 1993.

Rombach, H. Dieter. "Software Specifications: A Framework," ," Software Engineering Institute, Carnegie Mellon University, SEI Curriculum Module SEI-CM-11-2.1, January, 1990.

RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification," RTCA, Inc., December 1, 1992.

Rushby, John. "Critical Systems Properties: Survey and Taxonomy," Computer Science Laboratory, SRI International, Technical Report CSL-93-01, February, 1994.

Rushby, John. "Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems," Computer Science Laboratory, SRI International, January, 1992.

Serban, Cristina, and McMillin, Bruce. "From Formal Security Specifications to Executable Assertions – A Distributed Systems Preliminary Study," Department of Computer Science, University of Missouri-Rolla, CSC 95-01, April 27, 1995.

Serban, Cristina, and McMillin, Bruce. "Run-Time Security Evaluation (RTSE) for Distributed Applications," Department of Computer Science, University of Missouri-Rolla.

Shankar, Natarajan. "Mechanical Verification of a Generalized Protocol for Byzantine Fault-tolerant Clock Synchronization," Computer Science Laboratory, SRI International, January, 1992.

SSP 30309, Revision E. "Safety Analysis and Risk Assessment Requirements Document," National Aeronautics and Space Administration, International Space Station Alpha Program, Johnson Space Center, October 28, 1994.

SSP 50038, Revision B. "Computer-Based Control System Safety Requirements," National Aeronautics and Space Administration, International Space Station Alpha Program, Johnson Space Center, November 17, 1995.

Voas, Jeffrey; Charron, Frank; McGraw, Gary; Miller, Keith; and Friedman, Michael. "Predicting How Badly 'Good' Software can Behave," Reliable Software Technologies Corporation.

Weinstock, Charles B., and Gluch, David P. "A Perspective on the State of Research in Fault-Tolerant Systems," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-SR-008, June, 1997.

Wilde, Norman. "Understanding Program Dependencies," Software Engineering Institute, Carnegie Mellon University, SEI-CM-26, August, 1990.